Original published: M. Hofmann and G. Rünger. A partitioning algorithm for parallel sorting on distributed memory systems. In *Proceedings of the IEEE 13th International Conference on High Performance Computing and Communications (HPCC 2011)*, pages 402–411. IEEE, 2011. Online available at http://dx.doi.org/10.1109/HPCC.2011.59.

A Partitioning Algorithm for Parallel Sorting on Distributed Memory Systems

Michael Hofmann* and Gudula Rünger
Department of Computer Science
Chemnitz University of Technology, Germany
Email: {mhofma,ruenger}@cs.tu-chemnitz.de

***Abstract*—Parallel sorting methods for distributed memory systems often use partitioning algorithms to prepare the redistribution of data items. This article proposes a partitioning algorithm that calculates a redistribution specified by the number of data items to be finally located on each process. This partitioning algorithm can also be used for data items with weights, which might express a computational load to be expected, and to produce a redistribution with an individual accumulated weight of data items specified for each process. Another important feature is that data sets with duplicated data keys can be handled. Parallel sorting with those properties is often needed for parallel scientific application codes, such as particle simulations, in which the dynamics of the simulated system may destroy locality and load balance required for an efficient computation. It is applied to random sample data and to a particle simulation code requiring a sorting. Performance results have been obtained on an IBM Blue Gene/P platform with up to 32 768 cores. The results show that the proposed parallel sorting method performs well in comparison to other existing algorithms.**

***Keywords*-parallel sorting; data redistribution; particle simulations; load balancing; performance optimization;**

I. Introduction

Sorting a sequence of data items is one of the fundamental problems in computer science and an essential part of many algorithms and applications. The data items to be sorted need to have keys assigned on which a linear order is defined. Sorting a given sequence of data items is then the process of rearranging the data items into a new sequence with increasing (or decreasing) order of the keys. In distributed memory systems, the input and the output sequence of data items is distributed among a set of processes. In this case, parallel sorting involves communication to redistribute the data items. Parallel sorting algorithms can be based on merging or based on partitioning [1]. In contrast to merge-based sorting, which is often used in shared-memory systems, partition-based sorting is suitable for distributed memory environments, because it requires only a single redistribution of the data items.

Parallel sorting is often exploited in parallel scientific applications to achieve locality and load balance in distributed memory systems. Parallel particle simulation codes, for example, use sorting algorithms for rearranging particle data, whose keys are derived from the spatial location of the particles in the simulated domain. Sorting the particles according to these keys results in a data distribution in which particles that are close to each other in the physical domain are also close to each other in memory. An improvement of the locality of memory accesses can result, which can lead to a reduction of the communication costs for subsequent parallel computations. The distribution of particles among processes can also have a significant effect on the load balance of subsequent computations. Thus, the parallel sorting of the particle data can be essential for improving the performance of the entire particle code. To achieve good results, parallel sorting algorithms should offer a method to control how the data items are to be distributed among processes. The contribution of this article is to propose such a parallel sorting algorithm with the following properties:

- The parallel sorting algorithm is separated into several algorithmic steps which can be individually adapted to a specific application or platform. This article focuses on the partitioning step in which all processes cooperatively decide how the data items are to be redistributed.
- The distribution of data items among processes can be controlled by specifying lower and upper bounds for the positions at which the globally sorted sequence should be split into sub-sequences for the processes. This can be used to create both homogeneous and inhomogeneous distributions of data items among processes. Inhomogeneous distributions, for example, may be useful for heterogeneous architectures for which data items need to be distributed among processors proportionally to their individual computational speed.
- A specific feature of the partitioning algorithm is that the distribution of data items among processes can be specified with respect to application-specific weights that are assigned to the data items. The weights can be chosen, for example, according to the individual computational load caused by each data item. Thus, by creating a distribution of data items among processes with respect to these weights, the load balance of subsequent computations of an application can be improved.
- Also, the partitioning algorithm works for duplicated keys. The algorithm always provides the data distribution specified, even in cases in which all data items have

*Supported by the German Federal Ministry of Education and Research (BMBF) and by Deutsche Forschungsgemeinschaft (DFG).

the same key value or all data items are initially owned by a single process. This makes the parallel sorting algorithm suitable for the use in real-world applications that work on a-priori unknown data.

Partition-based parallel sorting algorithms usually exploit so-called *splitter keys* to describe how the data items need to be redistributed among the processes. However, to achieve the properties described above, the partitioning algorithm proposed uses *splitting positions* of the locally sorted sequences of data items. Splitting positions divide the local sequence of data items of each process into sub-sequences which are to be sent to other processes.

The algorithm is implemented on an IBM Blue Gene/P system. Especially for large numbers of processes, the parallel sorting method achieves lower runtimes in comparison to other existing algorithms. As example application, performance results of the parallel sorting algorithm within a parallel particle simulation code are shown.

The rest of this article is organized as follows. Section II presents related work. Section III introduces the partitioning algorithm proposed. Section IV shows performance results on a Blue Gene/P system. Section V concludes the article.

II. Related Work

Sorting is a common tasks in computer science and resulted in numerous contributions on sequential and parallel sorting [2], [3]. Recent research includes parallel sorting on GPUs [4], [5] as well as optimizations for multi-core architectures [6] and high scaling parallel environments [7].

Parallel sorting algorithms can be usually distinguished in partition-based algorithms and merge-based algorithms [1]. The algorithm proposed in this article is based on a partitioning algorithm and requires only a single data redistribution with standard all-to-all communication operations, thus benefiting from optimized communication libraries. In contrast, merge-based algorithms for distributed memory often require large numbers of communication steps [8].

Partition-based parallel sorting algorithms determine splitter keys by selecting random sample keys [9] or regular sample keys [10] from the data items. The selection of samples is repeated until splitter keys are determined that divide the data items into equal parts [9], [7]. However, application-specific load balancing metrics or heterogeneous environments can also require uneven distributions. Duplicated keys are handled by modifying the data items [11] or by adding counter values to the splitter keys [12]. In contrast to that, the partitioning algorithm proposed handles duplicated keys within a separate processing step that is performed only if it is inevitable for the solution requested.

Parallel radix sort methods separate the data items into buckets and assign entire buckets to parallel processes [13]. Refining large buckets is used to improve the load balancing of the parallel sorting [14]. This refinement can not proceed if buckets consists of duplicated keys only. Load Balanced Parallel Radix Sort splits buckets at process boundaries, but uses several all-to-all data redistribution steps [15].

Particle simulation codes use sorting to group particles into boxes or cells based on their spatial position within the simulation domain. This can be required for algorithmic reasons, such as collision detection [16], or for specific implementation strategies [17]. Furthermore, sorting particles is also used to optimize the cache usage [18] and as part of domain decomposition techniques [19]. Load balancing is performed as a separate step after the sorting [20], thus requiring an additional redistribution of the particle data. By sorting weighted data items as proposed in this article, the additional redistribution step can be omitted.

In [21], we described a library approach with parallel sorting algorithms for scientific computing and presented a merge-based parallel sorting algorithm with adaptive memory requirements. The use of parallel sorting algorithms in particle codes was discussed in [22]. This includes early results of the partitioning algorithm and optimizations for handling the data in the particle simulation code PEPC [23].

III. Parallel Sorting by Partitioning

In this section, the sorting problem as well as the partition-based parallel sorting algorithm with the partitioning algorithm are described.

A. Sorting Problem

Given a sequence $\mathcal{E} = \langle x_0, \ldots, x_{n-1} \rangle$ of n data items with keys on which a linear order "$\leq$" is defined, the **sorting problem** is to create a permutation $\mathcal{E}' = \langle x'_0, \ldots, x'_{n-1} \rangle$ of the input sequence $\mathcal{E}$ such that $x'_0 \leq x'_1 \leq \ldots \leq x'_{n-1}$ (i.e., $\mathcal{E}'$ is sorted). For distributed memory systems, the input sequence $\mathcal{E}$ is distributed among p parallel processes $q_1, \ldots, q_p$ such that each process $q_i, i \in \{1, \ldots, p\}$, owns a local sequence $\mathcal{E}_i = \langle x_{i,0}, \ldots, x_{i,n_i-1} \rangle$ of n_i data items, $\sum_{i=1}^{p} n_i = n$. The **parallel sorting problem** is to create a permutation (of the distributed input data items) consisting of the local sequences $\mathcal{E}'_i = \langle x'_{i,0}, \ldots, x'_{i,n'_i-1} \rangle, i = 1, \ldots, p$, $\sum_{i=1}^{p} n'_i = n$, such that the following conditions are met:

- **Intra process order:** Each sequence $\mathcal{E}'_i, i \in \{1, \ldots, p\}$, is sorted.
- **Inter process order:** The largest data item of sequence $\mathcal{E}'_i, i \in \{1, \ldots, p-1\}$, is not larger than the smallest data item of all sequences $\mathcal{E}'_z$ with $z > i$.

There exists a large variety of algorithm for solving sequential or parallel sorting problems [2], [3]. In the following, we propose a new parallel sorting algorithm based on a partitioning algorithm for data items with integer keys.

B. Overview of Parallel Sorting by Partitioning

Parallel sorting by partitioning consists of the following four steps:

I. **Local sort:** Each process $q_i, i \in \{1, \ldots, p\}$, sorts its local data items. For all following steps, $\mathcal{E}_i$ denotes the locally sorted sequence of data items of process q_i.

II. **Create a partitioning of the data items:** Each process $q_i, i \in \{1, \ldots, p\}$, divides its local sequence $\mathcal{E}_i$ into p sub-sequences $\mathcal{E}_{i,j}, j = 1, \ldots, p$, such that for each $j \in \{1, \ldots, p-1\}$ the largest data item

$$x^j_{max} = \max_{i=1,\ldots,p} \{x \in \mathcal{E}_{i,j}\}$$

of the j-th sub-sequences $\mathcal{E}_{i,j}$ of all processes $q_i, i = 1, \ldots, p$, is not larger than the data items of all sub-sequences $\mathcal{E}_{i,z}$ of all processes $q_i, i = 1, \ldots, p$, with $z > j$, i.e.:

$$x^j_{max} \leq x \text{ for all } x \in \mathcal{E}_{i,z}, i = 1, \ldots, p, z > j \quad .$$

This property is required to achieve the inter process order of data items defined in the previous section.

III. **Redistribute the data items:** Each process $q_i, i \in \{1, \ldots, p\}$, sends the data items of its j-th sub-sequence $\mathcal{E}_{i,j}$ to process $q_j, j = 1, \ldots, p$, and receives sub-sequences of data items from all other processes. This can be done by `MPI_Alltoallv` communication operations and efficient variants, see [24].

IV. **Local merge:** Each process $q_i, i \in \{1, \ldots, p\}$, rearranges the sub-sequences of data items received into the sorted local sequence $\mathcal{E}'_i$. Since the sub-sequences are already sorted, a sequential merge algorithm is used.

This article focuses on a partitioning algorithm for Step II.

C. Partitioning with Splitting Positions

Before the partitioning algorithm is described, we introduce the notation of *splitting positions*, which is used for the description of the partitioning of the data items.

Splitting Positions: Each process $q_i, i \in \{1, \ldots, p\}$, has a locally sorted sequence $\mathcal{E}_i = \langle x_{i,0}, \ldots, x_{i,n_i-1} \rangle$ of data items. The division of each local sequence $\mathcal{E}_i$ into p sub-sequences $\mathcal{E}_{i,j}, j = 1, \ldots, p$, can then be specified by a number of splitting positions $s_{i,j} \in \{0, \ldots, n_i\}, j = 0, \ldots, p$, with $s_{i,j-1} \leq s_{i,j}$ for $j = 1, \ldots, p$. If $s_{i,j-1} < s_{i,j}$, then sub-sequence $\mathcal{E}_{i,j}$ consists of data items $\langle x_{i,s_{i,j-1}}, \ldots, x_{i,s_{i,j}-1} \rangle$. Otherwise, sub-sequence $\mathcal{E}_{i,j}$ is empty. The first and last splitting positions of each process $q_i, i \in \{1, \ldots, p\}$, are fixed with $s_{i,0} = 0$ and $s_{i,p} = n_i$. Therefore, only $p \times (p-1)$ splitting positions need to be determined by the partitioning algorithm.

Using splitting positions to describe the partitioning of the data items represents an alternative approach to the commonly used *splitter keys*. When using splitter keys chosen from the keys of the data items it is only possible to divide the sequences of data items at positions where the data items have different keys. In contrast to that, splitting positions can divide the sequences of data items at any positions (even if all data items have equal keys).

The partitioning of the data items describes how the data items have to be redistributed between the processes for the partition-based parallel sorting. On each process $q_i, i \in \{1, \ldots, p\}$, splitting position $s_{i,j}, j \in \{1, \ldots, p-1\}$, separates the data items that have to be sent to the processes q_z with $z \leq j$ from the data items that have to be sent to the processes q_z with $z > j$. Because the positions of the data items are numbered consecutively starting by zero, the value of $s_{i,j}$ is equal to the number of data items that process q_i has to send to the processes q_z with $z \leq j$. The number of data items that process q_i has to send to one process q_j is given by the value $s_{i,j} - s_{i,j-1}$ for $i, j = 1, \ldots, p$.

Partitioning Boundaries: For each $j \in \{0, \ldots, p\}$, $S_j = \langle s_{1,j}, \ldots, s_{p,j} \rangle$ is denoted as a column of splitting positions that contains the j-th splitting position of all processes $q_i, i = 1, \ldots, p$. For each column $S_j, j \in \{0, \ldots, p\}$, the splitting positions are summed up over all processes to define a *partitioning boundary* $b_j = \sum_{i=1}^{p} s_{i,j}$. From the definition of the splitting positions it follows that

$$b_0 = 0, \quad b_p = n, \quad \text{and} \quad b_{j-1} \leq b_j \text{ for } j = 1, \ldots, p \quad .$$

The value of a partitioning boundary $b_j, j \in \{1, \ldots, p-1\}$, is equal to the total number of data items that all processes have to send to the processes q_z with $z \leq j$, and the value $b_j - b_{j-1}$ is equal to the total number of data items that all processes have to send to the process $q_j, j \in \{1, \ldots, p\}$.

Partitioning with Lower and Upper Bounds: The partitioning algorithm determines splitting positions such that each resulting partitioning boundary b_j is within a given interval $[b^j_{low}, b^j_{high}], j \in \{1, \ldots, p-1\}$. The lower and upper bounds b^j_{low} and $b^j_{high}, j = 1, \ldots, p-1$, have to fulfill the following conditions:

$$0 \leq b^j_{low} \leq b^j_{high} \leq n$$

$$b^{j-1}_{low} \leq b^j_{low} \text{ and } b^{j-1}_{high} \leq b^j_{high} \text{ for } j = 2, \ldots, p-1$$

and are chosen in advance by the user of the sorting algorithm to specify the load balance of the final result. The total number of data items that each process $q_j, j \in \{1, \ldots, p\}$, will finally own is equal to the value $b_j - b_{j-1}$. Since b_j is within the interval $[b^j_{low}, b^j_{high}]$ and b_{j-1} is within the interval $[b^{j-1}_{low}, b^{j-1}_{high}]$, it follows that process q_j will own at least $b^j_{low} - b^{j-1}_{high}$ data items and at most $b^j_{high} - b^{j-1}_{low}$ data items. Therefore, the minimum and maximum number of data items that each process should own can be controlled by specifying appropriate lower and upper bounds.

In this article, we introduce the explicit usage of individual lower and upper bounds for each partitioning boundary. This allows the creation of even and uneven distributions of data items among processes. Other parallel sorting algorithms usually support only even distributions [9], in some cases with a specific amount of imbalance allowed [7]. This can be considered to be a special case in our approach. To get a distribution of data items among processes where each process will own at most $\frac{n}{p} + n_{imba}$ data items, the following lower and upper bounds for $j = 1, \ldots, p-1$ can be used:

$$b^j_{low} = j\frac{n}{p} - \frac{n_{imba}}{2} \text{ and } b^j_{high} = j\frac{n}{p} + \frac{n_{imba}}{2} \quad . \qquad (1)$$

D. Determining a Single Column of Splitting Positions

This subsection describes how the processes cooperatively determine the splitting positions of one column $S_j, j \in \{1, \ldots, p-1\}$. The splitting positions of a column S_j depend on each other, because they have to be chosen such that the condition of Step II, Sect. III-B is achieved and the partitioning boundary b_j is within the interval $[b^j_{low}, b^j_{high}]$.

Searching for the splitting positions proceeds in several rounds and uses the binary digit representation of the integer keys of the data items. The data items have r-bit integer keys with bit positions $1, \ldots, r$ enumerated from the lowest (rightmost) to the highest (leftmost) bit. Each round of the search uses t bits of the keys ($1 \leq t \leq r$), starting with the highest t bits in the first round. Hence, at most $\lceil \frac{r}{t} \rceil$ search rounds need to be performed and in the last round fewer than t bits can remain to be used.

Determining Candidates for the Splitting Positions: In each search round, each process $q_i, i \in \{1, \ldots, p\}$, selects $2^t + 1$ possible candidates $\bar{s}^k_i, k = 0, \ldots, 2^t$, for its splitting position $s_{i,j}$. The selection of splitting candidates is done by considering the local sub-sequence $\langle x_{i,l_{i,j}}, \ldots, x_{i,h_{i,j}-1} \rangle$ of data items with $0 \leq l_{i,j} \leq h_{i,j} \leq n_i$. If $l_{i,j} = h_{i,j}$, then process q_i considers none of its data items. Splitting candidates are the positions that cut the sequence of r-bit words into 2^t sub-sequences, so that each sub-sequence contains the r-bit words with the same t-bit word at the bit positions considered in this round.

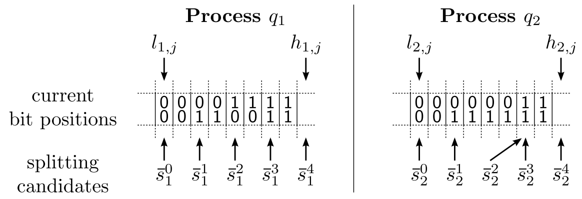

Figure 1. Example for the selection of splitting candidates with $p = 2$ processes in one search round. Each process uses $t = 2$ bits to select $2^t + 1 = 5$ splitting candidates.

Figure 1 shows an example with two processes for selecting splitting candidates. Each process uses $t = 2$ bits of the keys of the data items to select five splitting candidates from its local sub-sequence of data items. Process q_2 has no data items with 2-bit word 10. Therefore, $\bar{s}^2_2$ and $\bar{s}^3_2$ are equal to the position of the first data item with 2-bit word 11.

The sub-sequence of data items considered by each process changes during the algorithm. The search starts with $l_{i,j} = 0$ and $h_{i,j} = n_i$ in the first round (i.e., all data items are considered). In each search round, all processes select one of their 2^t sub-sequences (all processes select the same) to refine the search in the next round. The definition of the splitting candidates always leads to $\bar{s}^0_i = l_{i,j}$ and $\bar{s}^{2^t}_i = h_{i,j}$. Therefore, only $2^t - 1$ splitting candidates need to be determined. Since the local sequences of data items of each process are sorted, each splitting candidate can be determined using a binary search.

Compare Candidates with the Lower and Upper Bounds: When all processes have determined their splitting candidates in one search round, their values are summed up to create $2^t + 1$ boundary candidates $\bar{b}^k = \sum_{i=1}^{p} \bar{s}^k_i, k = 0, \ldots, 2^t$. The results of this summation are made available to all processes. These boundary candidates are compared with the lower and upper bounds b^j_{low} and b^j_{high} to continue the search in one of the two following ways:

- **Case A:** If there exists at least one $k \in \{0, \ldots, 2^t\}$ such that $b^j_{low} \leq \bar{b}^k_j \leq b^j_{high}$ (the smallest value of k is used if more than one exists), the search is finished successfully and each process $q_i, i \in \{1, \ldots, p\}$, uses $s_{i,j} = \bar{s}^k_i$ as splitting position for column S_j.
- **Case B:** If there exists $k \in \{1, \ldots, 2^t\}$ such that $\bar{b}^{k-1}_j < b^j_{low}$ and $b^j_{high} < \bar{b}^k_j$, then each process $q_i, i \in \{1, \ldots, p\}$, sets $l_{i,j} := \bar{s}^{k-1}_i$ and $h_{i,j} := \bar{s}^k_i$ and performs another search round.

Parameters r, t, and $[b^j_{low}, b^j_{high}], j = 1, \ldots, p-1$, are the same on all processes. Therefore, the selection of case A or case B and the corresponding value k depends only on the boundary candidates. Since the values of the boundary candidates are made available to all processes, it follows that all processes perform the search in a synchronized way.

E. Partitioning for Duplicated Keys

If there are no duplicated keys (i.e., all data item have different key values), then the search always ends successfully after at most $\lceil \frac{r}{t} \rceil$ search rounds. Otherwise (i.e., with duplicated keys), it is possible that case B is chosen for a column $S_j, j \in \{1, \ldots, p-1\}$, in the last search round, which would require to continue the search. However, since all bit positions have already been considered, all remaining data items to be considered have equal keys and no further splitting candidates can be selected. In practice, this situation mainly occurs, for example, if the number of data items with equal keys is very high or if the difference between the lower and upper bounds b^j_{low} and b^j_{high} is very small.

If the search is not finished successfully in the last search round, then a final round is appended. The goal of this final round is to select splitting positions $s_{i,j} \in [l_{i,j}, h_{i,j}]$ for each process $q_i, i \in \{1, \ldots, p\}$, such that b_j is within $[b^j_{low}, b^j_{high}]$. In this situation, the value $\left\lfloor \frac{1}{2}(b^j_{low} + b^j_{high}) \right\rfloor$ is chosen as target value for b_j. To achieve this target value, each process $q_i, i \in \{1, \ldots, p\}$, chooses its lowest possible value $s_{i,j} = l_{i,j}$ as starting point. Then, beginning at process q_1, the values of $s_{i,j}$ are increased until the target value of b_j is achieved. This procedure can be performed in parallel by letting each processes $q_i, i \in \{1, \ldots, p\}$, compute the increase required for itself as follows:

$$d_i = \left\lfloor \frac{1}{2}(b^j_{low} + b^j_{high}) \right\rfloor - (\sum_{z=1}^{p} l_{z,j} + \sum_{z=1}^{i-1} (h_{z,j} - l_{z,j}))$$

The value of d_i is the difference to the target value of b_j, when all processes q_z with $z < i$ choose their highest possible splitting position $s_{z,j} = h_{z,j}$ and all processes q_z with $z \geq i$ choose their lowest possible splitting position $s_{z,j} = l_{z,j}$. If $d_i < 0$, then no increase is required by process q_i. Otherwise, d_i is equal to the value by which $l_{i,j}$ has to be increased (but without exceeding $h_{i,j}$). The result of $\sum_{z=1}^{p} l_{z,j}$ is equal to the value of one boundary candidate from the last search round and is therefore already known to all processes. The partial summation $\sum_{z=1}^{i-1}(h_{z,j} - l_{z,j})$ can be computed with a prefix sum operation for all processes in parallel. After each process $q_i, i \in \{1, \ldots, p\}$, has determined its local value d_i, the splitting position $s_{i,j}$ is determined as follows:

$$s_{i,j} = \max\{l_{i,j}, \min\{l_{i,j} + d_i, h_{i,j}\}\}$$

F. Overall Search of All Splitting Positions

The partitioning algorithm determines all $p \times (p - 1)$ splitting positions together in an interleaved way. First, the search rounds using the highest t bit positions are performed for all columns of splitting positions. After that, the next lower t bit positions are used for all columns of splitting positions that need further search rounds. If the search ends successfully for one column of splitting positions, then this column is marked as finished and skipped in the following search rounds. The search continues until all bit positions have been considered or all columns are marked as finished. Interleaving the search rounds of several columns of splitting positions in this way allows a reuse of splitting candidates and boundary candidates that have already been determined.

Figure 2 shows the pseudo-code for the partitioning algorithm. The function PARTITION is executed by all processes in parallel in an SPMD way. Each process $q_i, i \in \{1, \ldots, p\}$, uses its locally sorted sequence $\mathcal{E}_i$ of data items as input and returns splitting positions $s_{i,j}, j = 1, \ldots, p-1$, as results of the partitioning algorithm.

After the initialization, the algorithm performs a loop executing the search rounds by iterating over the bit positions (l-loop in line 10). In each iteration of the l-loop, a loop over all columns of splitting positions that are not already marked as finished is performed (j-loop in line 11). During the j-loop, one search round is performed for each column S_j of splitting positions using the currently considered bit positions. This corresponds to the description in Sect. III-D.

In lines 12–14, the splitting candidates and boundary candidates are determined. The summation over all processes (line 14) is performed using a global reduction operation ALL-REDUCE-SUM that makes the result of the summation

```
1  /* executed by processes $q_i, i = 1, \ldots, p$, in parallel */
2  function PARTITION
3  input: locally sorted sequence $\mathcal{E}_i = \langle x_{i,0}, \ldots, x_{i,n_i-1} \rangle$
          of $n_i$ data items with $r$-bit integer keys
4  input: lower and upper bounds $b^j_{low}$ and
          $b^j_{high}, j = 1, \ldots, p-1$, of the boundary values
5  input: number of bits $t$ to use in a search round
6  output: splitting positions $s_{i,j}, j = 1, \ldots, p-1$
7  /* initialization */
8  set $l_{i,j} = 0$ and $h_{i,j} = n_i$ for $j = 1, \ldots, p-1$
9  /* search rounds for considering the key bits */
10 for $l = 1, \ldots, \lceil \frac{r}{t} \rceil$ do
11     for $j = 1, \ldots, p-1$ and column $S_j$ not finished do
12         for $k = 0, \ldots, 2^t$ do
13             determine splitting candidate $\bar{s}^k_i$ with a
                 binary search in sub-sequence
                 $\langle x_{i,l_{i,j}}, \ldots, x_{i,h_{i,j}-1} \rangle$ using the currently
                 considered bit positions
14             determine $\bar{b}^k = \sum_{z=1}^{p} \bar{s}^k_z$ using an ALL-
                 REDUCE-SUM communication operation
15         if (exists $k \in \{0, \ldots, 2^t\}$
16           with $b^j_{low} \leq \bar{b}^k \leq b^j_{high}$) then
17             /* Case A: search for $s_{i,j}$ finished */
18             set $s_{i,j} = \bar{s}^k_i$
19             mark column $S_j$ as finished
20         else
21             /* Case B: prepare next search round */
22             determine $k \in \{1, \ldots, 2^t\}$ with
                 $\bar{b}^{k-1} < b^j_{low}$ and $b^j_{high} < \bar{b}^k$
23             set $l_{i,j} = \bar{s}^{k-1}_i$ and $h_{i,j} = \bar{s}^k_i$
24 /* final round for handling duplicated keys */
25 for $j = 1, \ldots, p-1$ and column $S_j$ not finished do
26     determine $d_i$ using a PREFIX-SUM communication
         operation
27     set $s_{i,j} = \max\{l_{i,j}, \min\{l_{i,j} + d_i, h_{i,j}\}\}$
```

Figure 2. The function PARTITION implements the partitioning algorithm and is executed by p processes in parallel in an SPMD way. Each process $q_i, i \in \{1, \ldots, p\}$, uses its locally sorted sequence $\mathcal{E}_i$ of n_i data items as input and returns the splitting positions $s_{i,j}, j = 1, \ldots, p-1$.

available to all processes. After comparing the boundary candidates with the lower and upper bounds, either splitting position $s_{i,j}$ is found and column S_j is marked as finished (case A in lines 17–19), or the values of $l_{i,j}$ and $h_{i,j}$ are modified for the next search round (case B in lines 21–23).

After the l-loop is finished, the final round is performed by a loop over all columns of splitting positions that are still not marked as finished (line 25). This can only occur if the data items have duplicated keys and corresponds to the description in Sect. III-E.

All processes execute the algorithm in parallel in an SPMD way. Communication between the processes is only necessary for the global summation and the prefix sum operations (line 14 and line 26). Since these are blocking operations, this leads to a synchronisation between all processes. Since, the result of the global summation is available on all processes (and all other parameters are identical), it follows that all processes make the same decisions for the conditional statement in line 15. Therefore, all processes execute and terminate the algorithm synchronously. The runtime of the algorithm is dominated by the three nested for loops (line 10, line 11, and line 12) and the binary search inside the innermost loop (line 13).

G. Weighted Data Items

The partitioning algorithm shown in Fig. 2 can be modified to support the distribution of data items among processes with respect to weights associated with each data item. This requires alternative lower and upper bounds w^j_{low} and $w^j_{high}, j = 1, \ldots, p-1$, that refer to the weights of the data items. For example, to get a distribution of data items among processes such that the data items of all processes have equal accumulated weights, the lower and upper bounds can be determined with Eq. (1), but replacing the total number of data items n with the total weight of all data items.

To support weighted data items, the partitioning algorithm has to be modified in the following way: For each splitting candidate $\bar{s}^k_i, k = \{0, \ldots, 2^t\}$, on process $q_i, i \in \{1, \ldots, p\}$, the weights of all data items $x_{i,z}, z = 0, \ldots, \bar{s}^k_i - 1$ are accumulated (modification of line 13). After that, instead of the splitting positions, the accumulated weights are summed up over all processes with the ALL-REDUCE-SUM communication operation (line 14). The results of these summations are then compared to the alternative lower and upper bounds w^j_{low} and w^j_{high} to decide how the search continues (lines 15–23). The final round has to be modified analogously. Accumulating the weights requires additional iterations over the local data items of a process. Therefore, the modifications for weighted data items can increase the runtime of the partitioning algorithm.

IV. PERFORMANCE RESULTS FOR PARALLEL SORTING BY PARTITIONING

Performance results are shown for the partitioning algorithm, for parallel sorting by partitioning and for the use of the partitioning algorithm in a particle simulation code.

A. Experimental Setup

Performance results have been obtained on an IBM Blue Gene/P system [25]. A single compute node of the system consists of a 4-way SMP processor with 2 GiB main memory. The *virtual node mode* was used, leading to four processes on each compute node.

All parallel algorithms were implemented using the platform-specific *Message Passing Interface* library (MPI) of the Blue Gene/P system. This MPI library contains optimized collective communication operations [26]. The partitioning algorithm and the parallel sorting method use only collective communication operations and, thus, can benefit from these optimizations. The global summation (line 14 in Fig. 2) is performed with the `MPI_Allreduce` operation. The prefix sum operation (line 26 in Fig. 2) is performed with the prefix reduction operation `MPI_Exscan`. This collective operation is similar to the `MPI_Allreduce` operation, but the result on each process $q_i, i \in \{1, \ldots, p\}$, is accumulated only from the values of all processes q_z with $z < i$. The redistribution of the data items among the processes for the parallel sorting is performed with the `MPI_Alltoallv` operation.

The sample data items used for the experiments consist of 64-bit integer keys with random values in the range $[0, 2^{64}-1]$. Uniformly distributed random keys (with mean value 2^{63}) and normally distributed random keys (with mean value 2^{63} and standard deviation $\frac{1}{3}2^{63}$) were used. Additionally, non-uniform distributions of random keys were created by combining random keys with binary AND operations; this method was proposed in [27] and leads to distributions with large numbers of duplicated keys. All random values were created using the random number generators of the IBM Engineering and Scientific Subroutine Library (ESSL).

The lower and upper bounds b^j_{low} and $b^j_{high}, j = 1, \ldots, p-1$, of the partitioning algorithm were chosen according to Eq. (1). In the following, the maximum imbalance of a distribution of data items among processes denotes the largest difference between the number of data items a single process will own and the average number of data items all processes will own. Unless otherwise specified, $n_{imba} = 0.01\frac{n}{p}$ was used, which means that an even distribution of data items among processes with a maximum imbalance of 1 % (with respect to the average number of data items per process) is achieved.

B. Results for the Partitioning Algorithm

Performance experiments have shown that the number of candidate bits t used for the partitioning algorithm has a strong influence on its performance. Increasing the parameter t reduces the maximum number of search rounds ($\lceil \frac{r}{t} \rceil$ with $r = 64$ for 64-bit integer keys), but increases the number of splitting candidates ($2^t + 1$) and the communication required for the global reduction operations (line 14 in Fig. 2). For all following measurements, $t = 3$ was used, because small runtimes were achieved almost independent from the number of processes, from the number of data item, and from the distribution of the key values.

Figure 3 shows runtimes of the partitioning algorithm with and without weights depending on the number of processes using uniformly distributed random keys. The

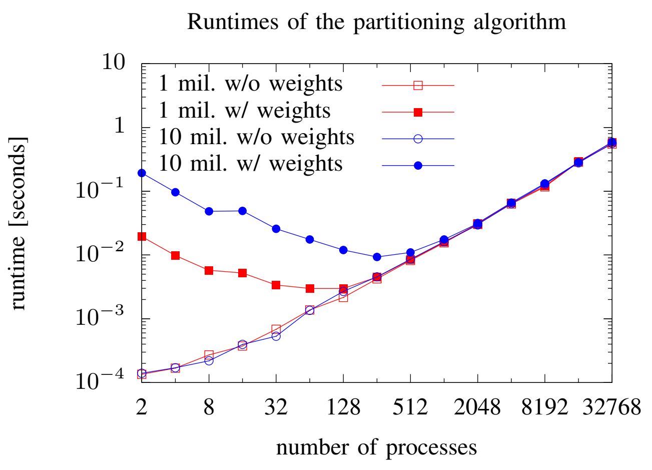

Figure 3. Runtimes of the partitioning algorithm depending on the number of processes with (w/) and without (w/o) weights using 1 mil. and 10 mil. data items in total.

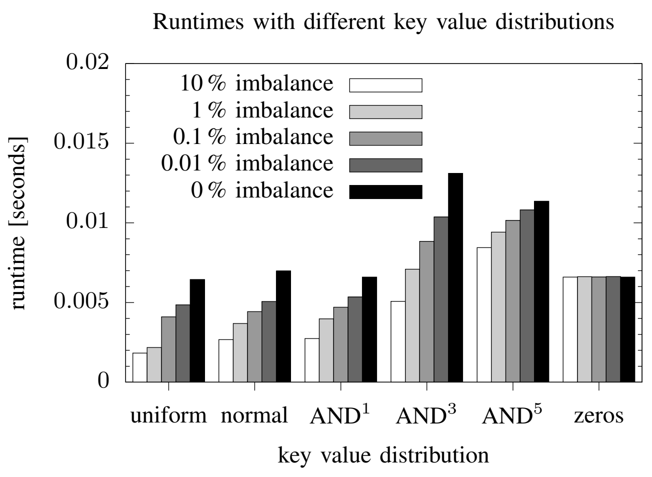

Figure 4. Runtimes of the partitioning algorithm with different maximum imbalance values using several random key value distributions and 128 processes. Each key value of the distributions AND^1, AND^3, and AND^5 is create with 1, 3, and 5 binary AND operations, respectively.

program version with weighted data items uses the same weight 1 for all data items. This leads to the same result as having no weights, but causes the runtime overhead for handling weighted data items. Without weights, the runtimes strongly increase for increasing numbers of processes, but are almost independent from the number of data items. This is the behavior expected, because the increasing number of processes also increases the number of splitting positions that need to be found. The number of data items has no significant influence on the runtime without weights, since determining the splitting candidates with binary search depends only logarithmically on the local number of data items and requires only a small part of the runtime. With weighted data items, the partitioning algorithm has significantly higher, but decreasing runtimes for small process numbers. This is caused by the linear iteration over the data items that is required to accumulate the weights. Hence, there is a benefit from the decreasing number of data items per process. The decrease in runtime stops when the local number of data items is too small. For large numbers of processes, the partitioning algorithms with and without weights have almost the same performance.

Figure 4 shows runtimes of the partitioning algorithm with different maximum imbalance values using several random key value distributions with 128 processes. Each key value of the distributions AND^1, AND^3, and AND^5 is create with 1, 3, and 5 binary AND operations, respectively. The results show that the maximum imbalance requested has a strong influence on the runtime of the partitioning algorithm. This is caused by the increasing number of search rounds that are required when the intervals $[b^j_{low}, b^j_{high}], j = 1, \ldots, p-1$, are small. Using key value distributions with large amounts of duplicated keys, which is the case for AND^3 and AND^5, leads to a significant increase of the runtime. In these cases, the final round has to be performed more often. With zero keys, it is impossible to find splitting positions with splitting candidates and therefore the final round is always performed.

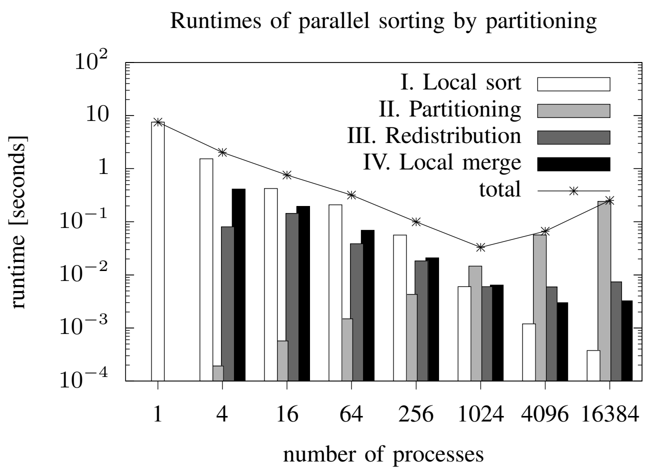

Figure 5. Runtimes of the four steps of parallel sorting by partitioning depending on the number of processes using 10 mil. data items in total.

C. *Results for Parallel Sorting by Partitioning*

The partitioning algorithm was used to implement the parallel sorting method described in Sect. III-B. Initially, the data items are evenly distributed among the processes. Local sorting in Step I is performed with a sequential radix sort algorithm as described in [21]. In Step II, all processes execute the partitioning algorithm from Sect. III-F. The redistribution of the data items in Step III is performed with the `MPI_Alltoallv` communication operation using the splitting positions as displacement values. Finally, in Step IV, each process merges the p sorted sub-sequences received using $p-1$ two-way merge operations. For the following experiments, uniformly distributed random keys were used.

Figure 5 shows runtimes of Steps I–IV depending on the

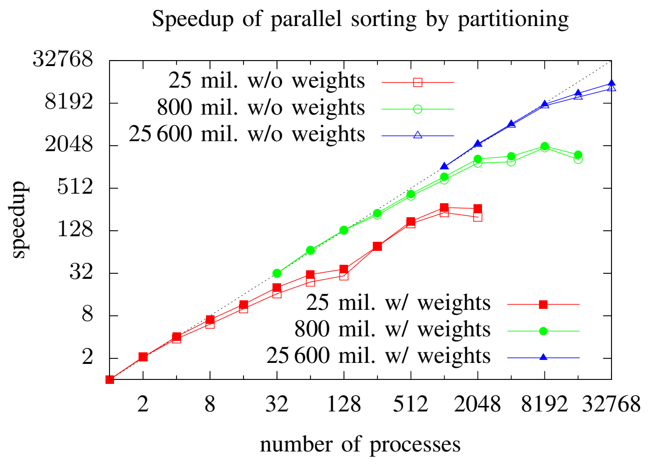

Figure 6. Speedup of parallel sorting by partitioning with (w/) and without (w/o) weights depending on the number of processes using 25 mil., 800 mil., and 25 600 mil. data items in total. Several results are not shown, because the number of data items per process was limited to 25 mil. and the measurements were stopped when no further decrease in runtime was achieved.

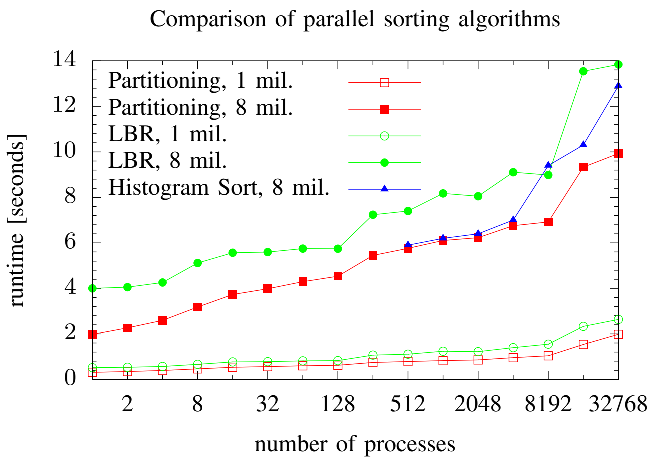

Figure 7. Runtimes of parallel sorting by partitioning, Load Balanced Parallel Radix Sort (LBR) [15], and Histogram Sort depending on the number of processes using 1 mil. and 8 mil. data items per process. Results for Histogram Sort are taken from [7].

number of processes using 10 mil. data items in total. The runtimes of the local sort step (I) and the local merge step (IV) decrease for increasing numbers of processes, which is caused by the decreasing number of data items per process. The runtime of the redistribution step (III) varies by an order of magnitude, but requires at most 19 % of the overall runtime, i.e., with 256 processes. The partitioning algorithm (II) shows the same behavior as in Fig. 3: The runtime increases for increasing numbers of processes. For large numbers of processes, the total runtime of parallel sorting by partitioning is dominated by the partitioning algorithm.

Figure 6 shows speedup values for parallel sorting by partitioning with and without weights depending on the number of processes using 25 mil., 800 mil., and 25 600 mil. data items in total. The program version with weighted data items uses the same weight 1 for all data items. Because of the limited memory of the compute nodes, the number of data items was limited to 25 mil. per process. The speedup values are calculated with respect to the runtime with 25 mil. data items per process. Depending on the total number of data items, parallel sorting by partitioning scales well up to 1024 processes, up to 8182 processes and even up to 32 768 processes. Superlinear speedup values are caused by cache effects in the local sort step (I). The speedup values with weights are higher, because with weighted data items the benefits from the decreasing number of data items per process have a stronger effect on the total runtime. However, the total runtime with weights is also higher, because of the additional overhead for the partitioning algorithm and for sorting the weighted data items locally.

Figure 7 compares runtimes for parallel sorting by partitioning, for Load Balanced Parallel Radix Sort (LBR) [15], and for Histogram Sort [7] depending on the number of processes using 1 mil. and 8 mil. data items per process. LBR creates an even distribution of data items among processes independent from the key values. It was implemented using 16 bits at once, thus performing four all-to-all data redistribution steps when sorting 64-bit integer keys. In comparison to parallel sorting by partitioning, LBR has higher runtimes. The high number of data redistribution steps lead to a large communication overhead (e.g., 70–80 % of the total runtime with 32 768 processes). When sorting data items in real-world applications, this can become even worse.

Histogram Sort is an optimized parallel sorting method for high scaling parallel systems and uses overlapping of communication and computation. In comparison to parallel sorting by partitioning, Histogram Sort achieves about the same runtime with up to 4096 processes. With more than 4096 processes, parallel sorting by partitioning is about 25 % faster than Histogram Sort. All three parallel sorting methods show a similar increase in runtime for large numbers of processes. For LBR and parallel sorting by partitioning, this increase is mainly caused by the data redistribution with the `MPI_Alltoallv` communication operation.

D. Results for Particle Data Sorting

As an example for an application that requires parallel sorting of weighted data items, we use an implementation of the Barnes-Hut algorithm for the calculation of long-range interactions in particle simulations [28]. The parallel implementation used is part of the freely available parallel tree code PEPC [23]. This application can be used to simulate the dynamical evolution of a system of particles by calculating the interactions between particles at discrete time steps (using the Barnes-Hut algorithm) and modifying their positions and velocities accordingly. The size of the particle systems depends strongly on the field of application and can range from thousands to even billions of particles.

The Barnes-Hut algorithm uses a hierarchical partitioning of the particles into boxes. The parallel implementation is based on the *Hashed Oct-Tree* scheme [17], where the position of a box is encoded in their box number such that the oct-tree of boxes is not stored as hierarchical data structure but as a hash table. The box numbers are recursively created from the coordinates of the boxes. The resulting linear ordering of the boxes corresponds to a Morton ordering, which retains the spatial locality of the boxes. All particles are assigned a 64-bit integer key according to the number of the box they are located in. By sorting all particles according to their keys, particles that are close to each other (in the particle system) become close to each other in memory. The parallel implementation uses parallel sorting to redistribute the particles among the processes such that particles that are close to each other are distributed to the same process (with high probability). In PEPC, each particle is assigned a weight value that approximates the computational load caused by that particle. By creating an even distribution of particles among processes with respect to these weights, the load balancing of the parallel implementation is improved.

The partitioning algorithm was used to implement the sorting of the particles. The particles are distributed among the processes such that a maximum imbalance of 1 % is achieved with respect to the weights of the particles. This new particle sorting method replaces the original method in PEPC, i.e., a recursive adaptation of Parallel Sorting by Regular Sampling [10] with weighted samples [19]. A more detailed description of the original and the new particle sorting can be found in [22]. Homogeneous and inhomogeneous distributions of particles were used. Inhomogeneous distributions were created according to the Plummer model, which means that the particles are clustered around a center. This leads to load imbalances during the computations.

Figure 8 shows runtimes with the original and the new parallel sorting method of PEPC depending on the number of processes using 6.4 mil. particles with an inhomogeneous distribution. With small numbers of processes, the particle data sorting requires only a small part of the total runtime. Additionally, the difference between the original and the new parallel sorting method is very small, because the runtime is dominated by the data redistribution step, which is similar in both methods. For increasing numbers of processes, the new parallel sorting method is about an order of magnitude faster than the original method. In these cases, the particle data sorting also requires a significant part of the total runtime of each time step. Thus, the reduction in runtime for the particle data sorting also leads to a reduction of the total runtime of each time step of about 10 % with 4096 processes.

Figure 9 shows runtimes of single time steps of PEPC (with the new parallel sorting method) with and without load balancing with weights depending on the number of processes. Homogeneous and inhomogeneous particle distributions with 25 000 particles per process were used. In these experiments, particle data sorting requires only about 1–2 % of the runtime of each time step. Thus, the differences between parallel sorting with weights and without weights are small in comparison to the overall runtime of each time step. However, using the weights to distribute the data items among processes during the parallel sorting leads to a significant reduction in runtime for the computations of a single time step of PEPC. The reduction in runtime is up to 8 % for homogeneous distributions of particles. Inhomogeneous distributions of particles lead to larger runtimes, but the reduction in runtime with weights is up to 18 %.

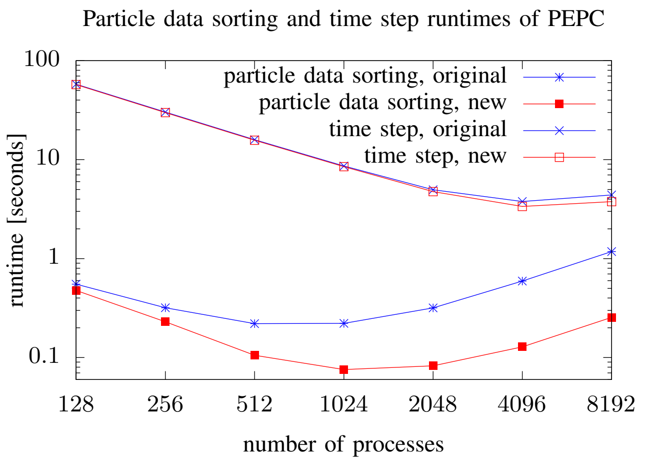

Figure 8. Runtimes with the original and the new parallel sorting method of PEPC depending on the number of processes using 6.4 mil. particles with an inhomogeneous particle distribution. Runtimes are shown for the particle data sorting within a single time step and for all computations of a single time step. Results with smaller numbers of processes are not shown, because the memory requirements could not be fulfilled.

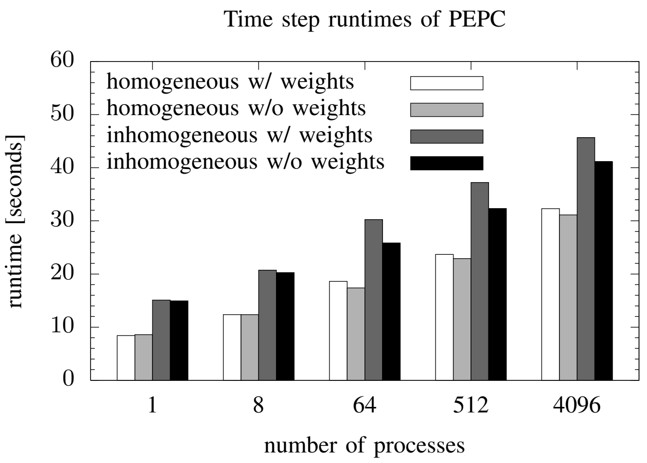

Figure 9. Runtimes of a single time step of PEPC with (w/) and without (w/o) load balancing with weights depending on the number of processes using 25 000 particles per process with homogeneous and inhomogeneous particle distributions.

V. Conclusion

In this article, we have proposed a novel partitioning algorithm to be used for a partition-based parallel sorting method for distributed memory systems. The sorting algorithm allows the user or the application code to control the distribution of the data items among the processes. Since weighted data items are supported, the partitioning algorithm is adaptable to application-specific load balancing metrics. Also duplicated keys are handle appropriately, so that any kind of sequence can be sorted. The performance results have shown that in real-world applications, the additional costs for handling weighted data items can be overcompensated by an improved load balancing in the scientific application. In summary, this leads to reductions of the runtime in comparison to existing parallel sorting methods, especially for large numbers of processes.

Acknowledgment

The measurements were performed at the John von Neumann Institute for Computing, Forschungszentrum Jülich, Germany. http://www.fz-juelich.de/nic

References

[1] Y. Manolopoulos, Y. Theodoridis, and V. Tsotras, *Advanced Database Indexing*. Kluwer Academic Publishers, 1999.

[2] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching (2nd ed.)*. Addison-Wesley, 1998.

[3] S. G. Akl, *Parallel Sorting Algorithms*. Academic Press, 1985.

[4] Y. Xiaochun, F. Dongrui, L. Wei, Y. Nan, and P. Ienne, "High Performance Comparison-Based Sorting Algorithm on Many-Core GPUs," in *Proc. of the 2010 IEEE Int. Symp. on Parallel & Distributed Processing*. IEEE, 2010.

[5] N. Leischner, V. Osipov, and P. Sanders, "GPU sample sort," in *Proc. of the 2010 IEEE Int. Symp. on Parallel & Distributed Processing*. IEEE, 2010.

[6] S. Hao, Z. Du, D. Bader, and Y. Ye, "A Partition-Merge Based Cache-Conscious Parallel Sorting Algorithm for CMP with Shared Cache," in *Proc. of the 2009 Int. Conf. on Parallel Processing*. IEEE, 2009, pp. 396–403.

[7] E. Solomonik and L. V. Kalé, "Highly Scalable Parallel Sorting," in *Proc. of the 2010 IEEE Int. Symp. on Parallel & Distributed Processing*. IEEE, 2010.

[8] A. Tridgell and R. P. Brent, "A General-Purpose Parallel Sorting Algorithm," *Int. J. of High Speed Computing*, vol. 7, no. 2, pp. 285–301, 1995.

[9] D. J. DeWitt, J. F. Naughton, and D. A. Schneider, "Parallel Sorting on a Shared-nothing Architecture Using Probabilistic Splitting," in *Proc. of the 1st Int. Conf. on Parallel and Distributed Information Systems*. IEEE, 1991, pp. 280–291.

[10] H. Shi and J. Schaeffer, "Parallel Sorting by Regular Sampling," *J. of Parallel and Distributed Computing*, vol. 14, no. 4, pp. 361–372, 1992.

[11] C. Cérin and J.-L. Gaudiot, "On a Scheme for Parallel Sorting on Heterogeneous Clusters," *Future Generation Computer Systems*, vol. 18, no. 3, pp. 353–372, 2002.

[12] D. R. Helman, J. JáJá, and D. A. Bader, "A New Deterministic Parallel Sorting Algorithm With an Experimental Evaluation," *ACM J. of Experimental Algorithmics*, vol. 3, 1998.

[13] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson, "High-Performance Sorting on Networks of Workstations," in *Proc. of the 1997 ACM SIGMOD Int. Conf. on Management of Data*. ACM, 1997, pp. 243–254.

[14] S.-J. Lee, M. Jeon, D. Kim, and A. Sohn, "Partitioned Parallel Radix Sort," *J. of Parallel and Distributed Computing*, vol. 62, no. 4, pp. 656–668, 2002.

[15] A. Sohn and Y. Kodama, "Load Balanced Parallel Radix Sort," in *Proc. of the 12th Int. Conf. on Supercomputing*. ACM, 1998, pp. 305–312.

[16] L. Dagum, "Data Parallel Sorting for Particle Simulation," *Concurrency: Practice and Experience*, vol. 4, no. 3, pp. 241–255, 1992.

[17] M. S. Warren and J. K. Salmon, "A Parallel Hashed Oct-Tree N-body Algorithm," in *Proc. of the 1993 ACM/IEEE Conf. on Supercomputing*. ACM, 1993, pp. 12–21.

[18] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl, "GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation," *J. of Chemical Theory and Computation*, vol. 4, no. 3, pp. 435–447, 2008.

[19] P. Gibbon, R. Speck, A. Karmakar, L. Arnold, W. Frings, B. Berberich, D. Reiter, and M. Maek, "Progress in Mesh-Free Plasma Simulation With Parallel Tree Codes," *IEEE Trans. on Plasma Science*, vol. 38, pp. 2367–2376, 2010.

[20] L. V. Kalé, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten, "NAMD2: Greater Scalability for Parallel Molecular Dynamics," *J. of Computational Physics*, vol. 151, no. 1, pp. 283–312, 1999.

[21] H. Dachsel, M. Hofmann, and G. Rünger, "Library Support for Parallel Sorting in Scientific Computations," in *Proc. of the 13th Int. Euro-Par Conf.* Springer, 2007, pp. 695–704.

[22] P. Gibbon, M. Hofmann, G. Rünger, and R. Speck, "Parallel Sorting Algorithms for Optimizing Particle Simulations," in *Proc. of the IEEE Int. Conf. on Cluster Computing, Workshops and Posters*. IEEE, 2010, pp. 1–8.

[23] "PEPC: A Multi-Purpose Parallel Tree-Code," http://www.fz-juelich.de/jsc/pepc.

[24] M. Hofmann and G. Rünger, "An In-place Algorithm for Irregular All-to-All Communication with Limited Memory," in *Recent Advances in the Message Passing Interface: 17th European MPI User's Group Meeting*, vol. 6305. Springer, 2010, pp. 113–121.

[25] IBM Blue Gene Team, "Overview of the IBM Blue Gene/P Project," *IBM J. of Research and Development*, vol. 52, no. 1-2, pp. 199–220, 2008.

[26] A. Faraj, S. Kumar, B. Smith, A. Mamidala, and J. Gunnels, "MPI Collective Communications on the Blue Gene/P Supercomputer: Algorithms and Optimizations," in *Proc. of the 17th IEEE Symp. on High Performance Interconnects*. IEEE, 2009, pp. 63–72.

[27] K. Thearling and S. Smith, "An Improved Supercomputer Sorting Benchmark," in *Proc. of the 1992 ACM/IEEE Conf. on Supercomputing*. IEEE, 1992, pp. 14–19.

[28] J. Barnes and P. Hut, "A Hierarchical $O(N \log N)$ Force-Calculation Algorithm," *Nature*, vol. 324, no. 6096, pp. 446–449, 1986.